Intro to Git

Pranav Minasandra
Max Planck Institute of Animal Behavior

September 29, 2022

Note 1: Throughout this document, commands are written in `typewriter-text`, and must usually be entered as they are in your command-line. Within the typewriter text, some things might need to be modified, and are indicated using `<>`, like so: `command1 option1 <you need to change this>`. Sometimes, commands are long and need to be written in separate lines. When I write these, commands will begin with a `$`, like so:

```
$ meaningof argument1 "Life" argument2 "The Universe" argument3 "Everything"
```

In such cases, you only enter the command after the `$`, but don't have to type the `$` itself.

Note 2: While this document might seem full of command line arguments, in reality, there are just the following ten `git` commands:

Command	Use
`config`	Sets up things at the beginning
`init`	Starts git up
`add` and `rm`	Add and delete files from tracking
`status`	Displays current status
`log`	Displays history
`branch` and `switch`	Creating and switching between branches
`merge`	Joins two branches
`restore`	Lets you go back to a previous point

---

1 Installation

Depending on your operating system, you can install `git` on your system using one of the following methods.

1.1 Linux

If your system is based on Debian, you can use `sudo apt-get install git`. If your system is based on some other flavour of linux, you can use your typical installation method.

1.2 Mac

You can use `brew install git` to install `git` on Mac. That's what the internet says. I have never used a Mac.

1.3 Windows

Git says that you can install git for windows. Please do this, and make sure you have the software GitBash installed. GitBash is the command-line for running `git` on Windows, and is where you will run all your commands. It should automatically be installed when you install Git for Windows from the link above.

2 Getting help

If you need git-related help after the session on September 29, 2022, you can use the following resources.

- `man git`
- `man gittutorial`
- Chacon & Straub, *Pro Git* (Apress Publications, 2014)

3 Preparing for the session

To prepare for the session, make sure you have a command-line working where the command

```
$ git --version
```

gives you a version number and not an error. Also, to simulate the process of writing code at high speed, we will be making and sharing small text-files.[1] Please come prepared with your best one-liners, limericks, and ASCII-art which will play the role of code.

Everything that follows from this point will be covered in the workshop.

---

4 Introducing yourself to `git`.

Run the following commands

```
$ git config --global user.name "U G Babaji"
$ git config --global user.email "babajiug@gmail.com"
```

making sure that you use your own name and e-mail in place of the placeholders.

5 Starting a `git` project

Projects version-controlled by git are called git repositories. Choose a folder of your choice, and navigate to it using the command `cd </path/to/folder>`. Once there, enter the command

```
$ git init
```

to start `git`-based version control in this folder. `git` should now ideally say something like

`Initialized empty Git repository in <your folder here>`.

[1] Since all code is, in the end, text.

6 Creating and commiting your first few files

Create two files with names and contents of your choice. Now type the command

```
$ git status
```

If all has gone according to plan, `git` will now say something like:

```
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    <your first file>
    <your second file>

nothing added to commit but untracked files present (use "git add" to track
    )
```

Now, you need to add files to `git` to be tracked (like it just told us). One way to do this is to add them all individually:

```
$ git add <file1> <file2>
```

But this would be a pain if you had many many files. Easier would be to do this:

```
$ git add .
```

where `.` stands for 'the entire current directory'. Your files have been added to what is called the 'staging area', a temporary modification-tracking space. At this point, if we run

```
$ git status
```

again, we will see something like

```
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   <your first file>
    new file:   <your second file>
```

At this point, we can tell `git` to commit all our changes like so:

```
$ git commit
```

Git should now prompt you to enter a message. A `git commit` can be thought of as "With this code, I have reached a point where I might want to share this progress with others, or a point I might want to come back to in the future."

Now, as an exercise, modify one of your files, and then add the modified file to staging, and commit your changes. Look at the output of `git status` before and every step.

To remove a file from your repository, you can use

```
$ git rm <filename>
```

Use this to remove a file, and then add and commmit yet another file with some contents to your repository. By this point, you should have committed three times. Run the command

```
$ git log --graph
```

and you must see something like

```
* commit d3c8c7b5ff81dae69770c39fe52a8b6a6973d8fb (HEAD -> master)
| Author: Pranav Minasandra <pminasandra@ab.mpg.de>
| Date:   Wed Sep 28 11:46:37 2022 +0200
|
|     Roman joke added.
|
* commit e191e2c3d5ddeb21012263aa4de14c93f35f88ec
| Author: Pranav Minasandra <pminasandra@ab.mpg.de>
| Date:   Wed Sep 28 11:44:32 2022 +0200
|
|     This is a useful comment.
|
* commit 429c1552d23be58b04a5b3e56892b6701b0bf243
  Author: Pranav Minasandra <pminasandra@ab.mpg.de>
  Date:   Wed Sep 28 11:40:04 2022 +0200

      This is a helpful message.
```

This might seem like an unhelpful message with random garbled text in it, but I assure you, it's not. We will see the full use of this stuff when we move on to:

7 Branching

Branching is useful when you want to *safely* try out an alternative approach to something. Imagine that reality itself has split into two alternating universes, and you have written different code in each. To create out first branch, let's use:

```
$ git branch <your-branch-name>
```

Note that `<your-branch-name>` must absolutely not contain spaces or special characters. Use - and _ if you have to.

Typing the following command will print out all the branches that exist

```
$ git branch
```

The output should look something like this:

```
  <your-branch-name>
* master
```

`master` is the name of the default branch in `git`, which was created when we first initialised this repository.[2] The `*` before `master` says that we are still on that branch. This is because while we created a separate branch, we still haven't moved to it. We move to the other branch by using

```
$ git switch <your-branch-name>
```

and `git branch` will show you that you have switched.

Now, choose a file and modify it. Add another new file with some content. Add and commit both these files. After this, carefully examine the contents of each file, before switching back to the master branch using

```
$ git switch master
```

[2] While git and things like GitLab call the default branch 'master', github has recently started referring to it as 'main'. This is extremely annoying.

and look at your files again. You will notice that the files have gone back to what they used to be, and the additional file you created in your separate branch has vanished.

8 Merging and merge conflicts

Remember the file you modified in your new branch? It hasn't been changed in `master`, because using our older analogy, we made the changes in a parallel universe. Modify this file again in `master`, but make different changes from the ones you made earlier. Now as it stands, you have two different versions of this file in two branches, and you also have an additional file in your branch that doesn't exist in `master`. All good so far? Good. It's time to make universes collide.

First, make sure you're on the branch `master` with the command `git branch` again.

Then, use the command

```
$ git merge <your-branch-name>
```

to bring the two universes together. You notice that `git` complains immediately:

```
Auto-merging <the-file-you-modified>
CONFLICT (content): Merge conflict in <the-file-you-modified>
Automatic merge failed; fix conflicts and then commit the result.
```

The first thing to notice is that your extra file from your branch has been brought over to `master`, no questions asked. But the file you modified is an issue, git does not know how to resolve the conflict between its two versions.

Go ahead and open the file, and you will see that `git` has helpfully annotated the conflicting lines for you. As an exercise, modify the file by choosing one of the modifications, and then add and commit everything to master again. We can see everything we've done in graphical representation now using

```
$ git log --graph
```

Let's now destroy the branch we created, we don't need it anymore since it has been merged with `master`.

```
$ git branch -d <your-branch-name>
```

9 Undoing changes

There are many ways of 'going back in time' using `git`. We will discuss mainly two of these.

9.1 Fixing recent errors

Suppose, after your most-recent commit, you modified a particular file and messed up everything. You can call the command

```
$ git restore <modified-file>
```

to revert that file back to its state at the most recent commit.

9.2 Going back to an old commit

Let's say you've been trying out a whole new approach, and you've created many new code files and in the end, have found out that the whole approach was incorrect, or wrongly implemented. Like you see in `git log`, each commit is accompanied by a commit ID, something like: `e191e2c3d5ddeb21012263aa4de14c93f35f88ec`

From `git log`, choose a previous commit that you want to go back to. We then create a different branch, a parallel universe which is basically this universe at a previous point in time.

```
$ git branch <branch-name> e191e2c3d5ddeb21012263aa4de14c93f35f88ec
```

If you are not very masochistic, you will not want to type out the entire commit ID. `git` lets you get away with this by typing the first few characters as well:

```
$ git branch <branch-name> e191e2c
```

should do.

There are far more permanent ways of undoing changes than this. However, these methods make you prone to accidentally losing huge amounts of progress. Let us therefore not speak of such blasphemous things, and content ourselves with the `git branch` based method above.

As a quick exercise, modify one of your files, and then restore it to its state during the previous commit. After this, make a branch from your very first commit, switch to it, and ponder about how much `git` you've learned over the last hour.